PyFX: A framework for programming real-time effects

Calle Lejdfors and Lennart Ohlsson

Department of Computer Science, Lund University
Box 118, SE-221 00 Lund, SWEDEN

{calle.lejdfors, lennart.ohlsson}@cs.lth.se

Abstract

Programming real-time effects for contemporary GPUs requires writing shader programs to run on the GPU as well as code for the render state setup logic performed by the CPU. While the GPU parts are well supported by high level programming languages, the effect frameworks commonly used for the CPU parts are lacking both in functionality and expressive power, which makes them difficult to work with.

In this paper we present an effect framework implemented as an embedded language in Python. We show that this high-level language for effect descriptions provide increased expressivity, without sacrificing declarativity of other frameworks. We show how some additional functional features, image-processing and off-screen render targets, cooperate with the effect language giving a rich environment for experimenting with both functional and expressive features of effect programming.

1 Introduction

Special effects in cinematic graphics have long relied on procedural techniques and in the last few years the evolution of graphics processors have made these techniques available for use in real-time graphics as well. Programming real-time effects is today significantly harder than programming cinematic effects. The reason is partly that the real-time constraints makes the problem harder simply because limits on the execution time implies restrictions on the algorithms that can be used. Another reason is that the tools and techniques available are not yet as mature and this makes the development process harder than it has to be.

In the world of cinematic graphics programming visual effects is known as shader programming. The term shader programming is also used in the context of real-time effects, but here it is commonly used to refer only to the part of the effect which is executed as a program on the GPU. We therefore use the term *effect programming* for the process of creating the complete real-time effect, including the shaders for the GPU but also code to be run on the CPU.

Real-time shaders can today be programmed in a number of high-level languages. NVIDIA's Cg [Mark et al., 2003], Microsoft's HLSL [Gray, 2003] and the OpenGL Shading language, GLSL, [ogl, 2002] are all based on the Renderman Shading Language, the established standard for programming cinematic effects. They have come a long way towards

being a flexible and efficient development tool for the GPU parts of an effect. Programming the CPU part of the effects, however, has still very limited support. Loading shaders to the GPU, binding their run-time parameters, setting pipeline states and controlling the execution of multiple passes are commonly done in application specific code and not integrated with the rest of the definition of the effect.

The current approach to effect programming is the use of *effect frameworks*. An effect framework handles pipeline state manipulation including downloading shader programs and textures, doing parameter passing from the application to the shader program, and for orchestrating multiple passes of an effect. It provides some facility for loading effects and shader programs, typically based on a text file format in which shader programs, pipeline states and pass specifications are listed.

The effect framework idea was first used in the Quake 3 engine [amd Brian Hook, 1999] to allow user scripting shaders, or what we would call effects, to control visual appearance of in-game characters and objects. The Q3 framework predates widely available programmable shader hardware and therefore lack many of the features of current effect frameworks. The two effect frameworks which are the most widely used are DirectX FX and CgFX which are extensions of the shader programming languages HLSL and Cg, respectively. Both these frameworks provide the ability to specify real-time effects using multiple shader programs and passes using a special file format syntax.

Although current effect frameworks improve the encapsulation of the GPU part and the CPU part of an effect, they still suffer at least two major drawbacks. The most critical one is that they lack some important features such as render to texture and image processing which are necessary for writing many of the effects used in modern graphics applications. Second, the syntax of current frameworks is rather restricted, *e.g.* there is no support for expressing abstraction or repetition, which many times can make the writing of effects tedious and error prone. Limited facilities for sharing common parts of effects result in redundancy and code duplication.

In this paper we present PyFX, an effect framework based on the Python [Python langage,] programming language. We show how an effect framework can be embedded in a very high level general purpose language enable easy development of new extension, in particular off-screen rendering and image processing. Furthermore embedding allows language constructs currently not supported by DXFX or CgFX, such as function definitions, classes, conditionals, and loops, to be used in the construction of effects. This enables design techniques established for other kinds of software to be applied to effect programming as well.

This paper is organized as follows. In Section 2 we present some related work. Section 3 gives an overview of current effect frameworks. In Section 4 we present PyFX followed by some examples (Section 5) which emphasize the advantages of PyFX. Finally we conclude with a discussion (Section 6).

2 Related work

Shader programming was started in 1984 when Cook introduced shade trees [Cook, 1984]. This represented a move away from the fixed function nature of previous systems to an interpreted model giving much greater flexibility for writing visual effects. In 1990 this gave rise to the RenderMan [Hanrahan and Lawson, 1990] shading language which became an industry standard for writing off-line shaders. Several propositions on how the power

and flexibility of off-line shader programming systems can be transferred to the world of real-time graphics have since been put forward.

One direction is taken by Olano and Lastra [Olano and Lastra, 1998], who describe a RenderMan-like real-time shading language for the PixelFlow system [Molnar et al., 1992]. This system consists of a SIMD array of general purpose processors for which shader programs are compiled via C++ and executed. While this system is well-suited for writing real-time shaders it bears little resemblance to the architecture of current GPUs.

Peercy *et al.* [Peercy et al., 2000], present a system for compiling RenderMan programs to multipass rendering on using an OpenGL 1.2 implementation extended with imaging support, high-precision data types (16 bit floating point), and dependent texturing. The key realization is that the graphics pipeline can be used as a SIMD processor where different OpenGL states correspond to different SIMD instructions operating in parallel on a set of fragments. A restricted version, called ISL, of the RenderMan language, which can run on top of any OpenGL 1.2 implementation, is also presented.

The realization of Proudfoot *et al.* [Proudfoot et al., 2001] that shading computations are carried out at different frequencies lead to a language which maps well to present day GPUs. The computational frequencies isolated where constant, per-group, per-vertex, and per-fragment. The compiler can use frequency information of a computation in order to map it to a particular stage of programmable pipeline.

Today a number of shader languages have found widespread use in the industry. These are HLSL [Gray, 2003] by Microsoft and Cg [Mark et al., 2003] by NVIDIA which are both very similar. The OpenGL Shading Language [ogl, 2002] achieved ARB approval in 2003 and is included in the OpenGL 2.0 specifications [Segal and Akeley, 2004]. All these languages are based around explicit separation of per-vertex and per-fragment computations (cf. Proudfoot *et al.*) and follow the uniform and varying data classification introduced in RenderMan.

On the consumer side, games such as Quake3 by ID Software makes heavy use of multipass multi-texture algorithms. The Q3 shader [amd Brian Hook, 1999] format (here called an *effect* format) provides a specialized language for controlling blending state, texture generation, fogging and texture application mode of multiple textures. The engine can then use, if available, multiple texture units to reduce the number of texture application passes needed. The format marks a first step in effect programming but it does not provide enough control be useful in a more general context. In particular, since Quake3 shaders predates consumer-level programmable graphics hardware, it does not support vertex or pixel shader programs.

Following in the footsteps of Quake3's shader format Microsoft introduced, coincident with the first generation of programmable hardware, the DirectX Effects [DirectX Effects,] (abbreviated here as DXFX). DXFX provide a large superset of the interface introduced by Q3 shaders allowing for vertex and pixel shader programs, stencil-, alpha-, and depth buffer operations, multiple passes, *etc.* to be used in the description of a visual effect. CgFX [CgFX 1.2 Overview,] was introduced together with the Cg language and provide an implementation of DXFX for a larger variety of platforms.

A related approach was taken by Lalonde and Schenk [Lalonde and Schenk, 2002] with the EAGL framework. This framework provides a portable method of describing the association of render methods (shader programs and render state setup annotated data binding semantics) and art assets, typically triangle meshes, allowing for efficient rendering on a number of platforms (PC/XBOX, Playstation 2, and GameCube). An off-line compiler takes combinations of render methods and art assets and generates platform specific

representations which can be efficiently rendered by the runtime system. Contrary to current effect frameworks, EAGL does not provide support for writing cross-platform render methods, every render method used must be reimplemented for every platform used.

The Sh shading language [McCool et al., 2002] demonstrates that a shading language can be implemented as embedded language in C++. This embedding gives the shader developer access to high-level language features, such as classes, templates, functions, and user-defined types, to be used in the construction of shader programs. Sh also provide support for run-time construction and composition of shader programs [McCool et al., 2004]. However, although it is implemented in C++ it lacks facilities for abstracting and expressing the CPU parts of effects.

The Vertigo shading language [Elliott, 2004] approaches shader programming from a novel angle. It is implemented as an embedded language in Haskell [Jones, 2003] and uses pure functions, *i.e.* functions without state or side-effects, to model the stream-like nature of the GPU. This results in a clean high-level model for programming shaders for generative geometry.

3 Current effect frameworks

Current effects frameworks such as DXFX and CgFX provide a number of features which simplify programming real-time visual effects. DXFX and CgFX extend HLSL and Cg, respectively, with the ability to

- declare effect variables which can be either user-editable, or *tweakable*, for data such as textures or colors, or engine internal, so called *non-tweakables*, for engine specific data such as transformation matrices.
- declare different implementations of an effect suitable for different platforms. Each of these so called *techniques* list a number of passes with each pass containing the render pipeline states to set before requesting the application to submit geometry to the render pipeline.

Both frameworks rely on effect specifications which are stored in text files and loaded at run-time. The effect file lists the variables, shader programs, techniques and passes making out the effect. Effect variables can optionally be annotated with application specific data such as the valid range of a parameter or the default filename of a texture to facilitate integration with, for instance, GUI development tools. Variables may also have an associated semantic, consisting of a string identifier, which can be used by the application to provide data independent of variable name, providing an abstraction when passing data to the effect.

Effects, being implemented in terms of external text files, can be changed without requiring recompiling the application. Using a standard set of annotations and semantics, introduced in DirectX 9, the format provides an application independent mapping of application data to effect data.

However, the file format is closed and the frameworks can not easily be extended with new functionality. Syntactic-wise the effect format allow the use of C-style preprocessor for writing simple syntactical extensions. Together with the possibility to group states in so called state blocks this provides a basic form of abstraction. On the downside, the preprocessor based approach lack the most basic abstraction and data-hiding functionality making it difficult to use effectively. Furthermore functionality such as loop-

ing and conditionals are also missing, requiring generative effects and effects containing hardware-specific implementations to rely on effect-specific application level code. For a more in-depth review of the problems associated with current effect frameworks we refer to [Lejdfors and Ohlsson, 2004].

4 PyFX

In this section we present our effect framework, PyFX, which is implemented as an embedded language in Python. Its main purpose is to be a tool for investigating which features and characteristics that are useful and desirable for effect programming. The fact that PyFX is embedded in a fully fledged programming language immediately makes it easier to write effects since it allows the use of all the language features from the host language. Using function definitions, loops, conditionals and modules to express and share common parts, the description of an effect becomes shorter and more clear.

The features in PyFX include those found in the DXFX and CgFX frameworks and in addition it provides:

- *Render-to-texture* – The framework can render to off-screen area which can be used as a texture in later stages of the effect or by another effect entirely.
- *Image processing support* – GPU based image processing operations can be applied to any texture or off-screen area.
- *Support for shader interfaces* – PyFX enables easy use of Cg's interfaces allowing runtime construction and composition of shader programs.

PyFX is built on top of OpenGL. It is designed to be independent of shader language and it currently supports Cg and GLSL. The implementation and application level interface of PyFX is described in more detail in [Lejdfors and Ohlsson, 2004].

4.1 Overview

The basic building block in an effect in PyFX is a "processing step" which is a generalization of the notion of a pass in other effect frameworks. Each step may or may not require the application to send geometry to the GPU. Currently there are two types of processing steps:

- *RenderGeometry* – these are the usual pass of other effect frameworks. Sets up the appropriate states and then instructs the application to transmit geometry.
- *ProcessImage* – used to perform 2D image processing between two images (which may reside in either textures, off-screen areas or the current screen buffer). It supports floating point target and source images/buffers allowing HDR image processing.

In addition to these functional features, the framework also provide, through language embedding, a complete programming language in which effects can be expressed. This has several benefits for effect programmers since it enables the use of common software design methodologies, such as abstraction and sharing in the construction of an effect.

This allows the effect writer to express an effect in a clear, to-the-point manner making development and debugging easier.

Every aspect of the framework is implemented as a class allowing easy extension and specialization. Together with using embedding in a high-level language, this enables engine and framework writers to experiment with new features with minimal impact on the rest of the framework.

5 Examples

As an example of the features common to PyFX, DXFX, and CgFX we will present an effect for doing bump-mapping using a normal map [Blinn, 1978]. While PyFX supports both Cg and GLSL only Cg will be used for example code.

The bump mapping effect uses a vertex program to translate the surface normal to tangent space. The fragment program then manipulates the normal using a normal map and a scaling parameter. The resulting normal is then used for shading computations. The initial part of the fragment shader program is shown below:

```
float4 fs(float2 texcoord : TEXCOORD0,
          //normal in tangential frame
          float3 normalT  : TEXCOORD1,
          ..., // parameters needed for lighting
          uniform sampler2D NormalMap,
          uniform float Scale) : COLOR
{
  float3 dN = tex2D(NormalMap, texcoord);
  normalT += Scale*(dN*2-1)
  normalT = normalize(normalT);
  // compute and return color ...
}
```

Listing 1.1: Bumpmapping in Cg

In contrast to DXFX and CgFX where shader code is mixed with parameter declarations, PyFX uses wrapper functions and classes to indicate which parts of the effect are shader programs and which are parameters *etc*. The wrapper for a Cg shader program is called `Cg` and is used as follows.

```
bumpmap = Cg(""" code as in Listing 1.1 """)
```

The triple-quotes `"""` are used by Python for multi-line string literals. Next we list the parameters of the effect together with their default values.

```
Scale = 0.2
NormalMap = Texture("default_normalmap.png")
```

Now we are ready to define the technique of this effect. In PyFX it consists of a single render step which uses `bumpmap` vertex and fragment programs.

```
technique = [RenderGeometry(
                VertexShader = bumpmap.vs(),
                FragmentShader = bumpmap.fs())]
```

This effect does not use any of the special features of PyFX and we could just as well have written it in CgFX or DXFX. The result of doing so would have been more code since we have to explicitly declare every parameter, including non-tweakables, used by the shader programs. In PyFX common variables, such as transformation matrices *etc.*, are passed implicitly to the shader program. This is described in detail in [Lejdfors and Ohlsson, 2004].

5.1 Generative effects

Because of the language embedding in Python we can use, for instance, repetitive elements and function abstractions, when writing our effects. Consider an effect for rendering furry objects. Such an effect can be achieved by rendering the object surrounded by a number of shells where the transparency of each shell increases with the distance to the object. This gives the impression of fur with decreasing thickness with increasing distance from the object [Lengyel et al., 2001]. Assuming we have a `FurShellShader` program for rendering a single fur shell `shell` of the object, we can describe a step for rendering fur-shells by the following constructor function:

```
def RenderFurShell(s):
    shell = s/FurThickness
    return RenderGeometry(
              AlphaBlendEnable = True,
              SrcBlend = SRCALPHA,
              DestBlend = ONE,
              VertexShader = vs(Shell=shell),
              FragmentShader = fs(Shell=shell))
```

Drawing the complete furry object using `NumberOfShells` shells amounts to rendering the solid object followed by rendering each fur shell, which is done using the following code:

```
technique = [RenderGeometry()] + \
            [RenderFurShell(i)
               for i in range(1,NumberOfShells)]
```

The use of abstraction and high-level constructs allow us to describe the fur effect succinctly. Key parameters such as the number of shells used or fur thickness can easily be changed without modifying other parts of the effect.

The same fur effect expressed in CgFX or DXFX would be much longer and more difficult to read since those formats lack a notion of repetition. With each shell render step written explicitly it is, for example, more difficult to change the number of shells used.

5.2 Image processing

A simple widely-used example of image processing is the glow effect [James and O'Rorke, 2004] used to simulate the nimbus due to atmospheric scattering which appear around brightly lit surfaces. It works by rendering an object to the screen, rendering the glowing parts of the object to an off-screen buffer, blurring the off-screen buffer and then additively blending the result to the screen. To express this in PyFX we start by introducing some helper functions for rendering the glow regions, blurring a buffer and additively blend some buffer onto some buffer.

```
def RenderGlowRegions(target):
    return RenderGeometry(
              Target=target,
              VertexShader=glowMask.vs(),
              FragmentShader=glowMask.fs())

def GaussianBlur(source):
    ...

def AdditiveBlend(source, target):
    return ProcessImage(Source=source,
```

```
                    Target=target,
                    SrcBlend = SRCALPHA,
                    DestBlend = ONE)
```

The technique which performs blurring can now be written simply as

```
technique = [RenderGeometry(),
             RenderGlowRegions(blurBuffer),
             GaussianBlur(blurBuffer),
             AdditiveBlend(blurBuffer, Screen)]
```

The result is a readable specification of what the effect does and how it does it.

Writing this effect in DXFX or CgFX is currently not possible without using application-specific workarounds.

Sharing common code

If we have two different effects which both do blurring, it makes sense to factor out this common part and describe it only once. We can, for example, define a function `BlurPostProcess`, contained in a module `Blurring`, by

```
def BlurPostProcess(source,target):
    return [RenderGlowRegions(source),
            GaussianBlur(source),
            AdditiveBlend(source, target)]
```

Now our original glow effect can be implemented as

```
import Blurring

blurBuffer = ...

technique = [RenderGeometry()] + \
   Blurring.BlurPostProcess(blurBuffer, Screen)
```

The other effect is obtained by the replacing `technique` by some other step sequence followed by the blur post-processing operation.

```
technique = [...] + \
   Blurring.BlurPostProcess(blurBuffer, Screen)
```

Using Python's modules we can share code between multiple effects simplifying the construction of many effects.

5.3 Supporting shader interfaces

The Sh shader language provides support for combining shader programs at run-time. Cg provide a similar method through the use of so called *interfaces*, similar in concept to interfaces in the Java programming language [Joy et al., 2000].

Prior to the introduction of interfaces the programmer was required to write one shader program for every combination of material and light-source. This results in a combinatorial explosion of the number of shader programs needed in an application. Interfaces provide us with a mechanism for abstracting implementation of a part of a shader program from its usage pattern. As an example, we can use the following Cg interface for a light source.

```
interface LightI {
  void intensityAt(in float3 position,
                   out float3 lightDirection,
                   out float3 color);
};
```

An implementation of a light source must be able to, given a point, return the color and direction of the incident light at that point. A program can use the `LightI` interface as:

```
void main(in float3 position,
          in float3 normal,
          out float3 color,
          uniform LightI Light)
{
  float3 L, C;
  light.intensityAt(position, L, C);
  color = //compute color
}
```

Listing 1.2: Using interfaces in Cg

Using this in PyFX we wrap up the program above in a Cg wrapper as:

```
fs = Cg(""" code as in Listing 1.2 """);
```

We can implement a number of different light sources which support the above interface. For instance a non-attenuated point light source can be implemented as

```
struct PointLight : LightI {
  float3 Position;
  float3 Color;

  void intensityAt(in float3 position,
                   out float3 lightDirection,
                   out float3 color)
  {
    lightDirection = normalize(position - Position);
    color = Color;
  }
};
```

Following this general outline we can easily implement spotlights, lights with attenuation, cube-mapped lights *etc.* We put all the light definitions in a `Lights` module.

```
pointLight = CgIImpl(""" code as above """);
spotLight = CgIImpl(""" ... """)
cubeMappedLight = CgIImpl(""" ... """)
```

Interface implementations are wrapped in PyFX `CgIImpl`-wrappers to indicate that they are not complete programs, only implementations of interfaces which are not executable in their own right.

Using the interfaces works just as variables so using the above program with a point light located at $(0, 100, 0)$ having red color is just

```
myPointLight = pointLight(Position=(0,100,0),
                          Color=(1,0,0))

technique = [RenderGeometry(
                FragmentShader =
                   fs(Light = myPointLight), ...)]
```

Changing the light source amounts to changing a single line in the effect file. The framework also support changing the light source at run-time, allowing flexible shader program composition to be used as an integral part of an application.

6 Discussion and conclusions

We have presented an effect framework which improves on current frameworks in two respects. By being an embedded language it can freely utilize the features of its host language. In this respect PyFX is similar to Sh and Vertigo, although these systems have slightly different focus. However, when the language embedding is done a low level language like C or C++, the power of host language features come at the cost of sacrificing the declarative style of DirectX FX and CgFX. Due to the high level character of Python, PyFX is able to provide the best of both worlds. It is both declarative and has a rich set of language features.

Furthermore, PyFX provides support for render to texture and image processing, features which are needed to write many common effects but which are not supported by current frameworks. With access to the source code of an existing framework, these features would probably be fairly straightforward to implement, and it is even likely that they will appear in some future version. Our experience is however with a framework which is embedded in an flexible language such amendments can be added very easily. It is possible to have a very short turn-around time for adding or modifying a feature, get feedback from using it and then change it again. Since it is likely that the wish list for effect framework features will continue to grow for some time still, we believe that this agility makes PyFX a suitable platform for exploring and evaluating the design space of effect frameworks.

In summary, PyFX framework represents work in progress but it already provide a flexible environment for prototyping and experimenting with effects and effect frameworks alike. We hope to continue exploring methods for providing good programming environments for the border land of CPU/GPU interaction. Hopefully we will also be able to provide a solid ground for extending the framework to handle GPGPU algorithms as well as providing further functional additions.

Bibliography

[ogl, 2002] (2002). *OpenGL 2.0 Shading Language White Paper*. 3D Labs, 1.2 edition.

[amd Brian Hook, 1999] amd Brian Hook, P. J. (1999). *Quake III Arena Shader Manual*. Id Software Inc., 12 edition.

[Blinn, 1978] Blinn, J. F. (1978). Simulation of wrinkled surfaces. In *Proceedings of the 5th annual conference on Computer graphics and interactive techniques*, pages 286–292. ACM Press.

[CgFX 1.2 Overview,] CgFX 1.2 Overview. CgFX 1.2 Overview. `http://developer.nvidia.com/`.

[Cook, 1984] Cook, R. L. (1984). Shade trees. In *Proceedings of the 11th annual conference on Computer graphics and interactive techniques*, pages 223–231. ACM Press.

[DirectX Effects,] DirectX Effects. DirectX SDK Documentation. `http://msdn.microsoft.com/`.

[Elliott, 2004] Elliott, C. (2004). Programming graphics processors functionally. In *Proceedings of the ACM SIGPLAN workshop on Haskell*, pages 45–56. ACM Press.

[Gray, 2003] Gray, K. (2003). *DirectX 9 programmable graphics pipeline*. Microsoft Press.

[Hanrahan and Lawson, 1990] Hanrahan, P. and Lawson, J. (1990). A language for shading and lighting calculations. In *Proceedings of the 17th annual conference on Computer graphics and interactive techniques*, pages 289–298. ACM Press.

[James and O'Rorke, 2004] James, G. and O'Rorke, J. (2004). *GPU Gems*, chapter Real-Time Glow, page 816. Addison Wesley Professional, 1 edition.

[Jones, 2003] Jones, S. P., editor (2003). *Haskell 98 Language and Libraries*. Cambridge University Press. ISBN: 0521826144.

[Joy et al., 2000] Joy, B., Steele, G., Gosling, J., and Bracha, G. (2000). *Java™ Language Specification*. Addison-Wesley Pub Co, 2 edition.

[Lalonde and Schenk, 2002] Lalonde, P. and Schenk, E. (2002). Shader-driven compilation of rendering assets. In *Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, pages 713–720. ACM Press.

[Lejdfors and Ohlsson, 2004] Lejdfors, C. and Ohlsson, L. (2004). Pyfx - an active effect framwork. In *Linköping Electronic Conference Proceedings*. `http://www.ep.liu.se/ecp/013/006/`.

[Lengyel et al., 2001] Lengyel, J., Praun, E., Finkelstein, A., and Hoppe, H. (2001). Real-time fur over arbitrary surfaces. In *Proceedings of the 2001 symposium on Interactive 3D graphics*, pages 227–232. ACM Press.

[Mark et al., 2003] Mark, W. R., Glanville, R. S., Akeley, K., and Kilgard, M. J. (2003). Cg: a system for programming graphics hardware in a c-like language. *ACM Trans. Graph.*, 22(3):896–907.

[McCool et al., 2002] McCool, M., Qin, Z., and Popa, T. (2002). Shader metaprogramming. In Ertl, T., Heidrich, W., and Doggett, M., editors, *Graphics Hardware*, pages 1–12.

[McCool et al., 2004] McCool, M., Toit, S. D., Popa, T. S., Chan, B., and Moule, K. (2004). Shader algebra. In *To appear at SIGGRAPH 2004*, page 9 pages.

[Molnar et al., 1992] Molnar, S., Eyles, J., and Poulton, J. (1992). Pixelflow: high-speed rendering using image composition. In *Proceedings of the 19th annual conference on Computer graphics and interactive techniques*, pages 231–240. ACM Press.

[Olano and Lastra, 1998] Olano, M. and Lastra, A. (1998). A shading language on graphics hardware: the pixelflow shading system. In *Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, pages 159–168. ACM Press.

[Peercy et al., 2000] Peercy, M. S., Olano, M., Airey, J., and Ungar, P. J. (2000). Interactive multi-pass programmable shading. In *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, pages 425–432. ACM Press/Addison-Wesley Publishing Co.

[Proudfoot et al., 2001] Proudfoot, K., Mark, W. R., Tzvetkov, S., and Hanrahan, P. (2001). A real-time procedural shading system for programmable graphics hardware. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pages 159–170. ACM Press.

[Python langage,] Python langage. The Python language. `http://www.python.org/`.

[Segal and Akeley, 2004] Segal, M. and Akeley, K. (2004). Opengl 2.0 specification.